CSE 5600
Computer Systems Architecture

Multicores
(Shared Memory Multiprocessors)

1

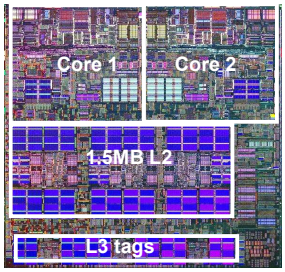
Multiplying Performance

- A single processor can only be so fast
  - Limited clock frequency
  - Limited instruction-level parallelism
  - Limited cache hierarchy
- What if we need even more computing power?
  - Use multiple processors!
  - But how?
- High-end example: Sun Ultra Enterprise 25k
  - 72 UltraSPARC IV+ processors, 1.5 GHz
  - 1024 GBs of memory
  - Niche: large database servers
- Modern example: AMD EPYC 9754
  - 128 Cores, 2.25 GHz



2

Multicore: Mainstream Multiprocessors

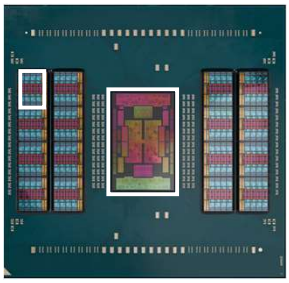


Why multicore? What else would you do with 1 billion transistors?

- **Multicore chips**
- **IBM Power5**
  - Two 2+GHz PowerPC cores
  - Shared 1.5 MB L2, L3 tags
- Intel Core i7 Quad
  - Four cores, private L2s
  - Shared 6 MB L3
- Sun Niagara
  - 8 cores, each 4-way threaded
  - Shared 2MB L2, shared FP
  - For servers, not desktop

3

Multicore: Mainstream Multiprocessors



Why multicore? What else would you do with 1 billion transistors?

- **Chiplet construction**
- 16 chiplets with cores
  - Eight 2.25 GHz cores each
  - Per-core 80 KB L1 cache
  - Per-core 1 MB L2 cache
  - Shared 512 MB L3 cache
- 1 I/O and interconnect chiplet
  - 12 DDR5 mem. controllers
  - Gen 5 PCIe

4

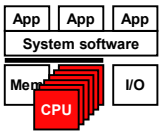
Application Domains for Multiprocessors

- Scientific computing/supercomputing
  - Examples: weather simulation, aerodynamics, protein folding
  - Large grids, integrating changes over time
  - Each processor computes for a part of the grid
- Server workloads
  - Example: airline reservation database
  - Many concurrent updates, searches, lookups, queries
  - Processors handle different requests
- Media workloads
  - Processors compress/decompress different parts of image/frames
- Desktop workloads…
- Gaming workloads…

But software must be written to expose parallelism

5

This Unit: Shared Memory Multiprocessors



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models

6

Identifying Parallelism

Consider

```
for (I = 0; I < 10000; I++)
    C[I] = A[I] * B[I];
```

or

```
struct acct_t { int balance; };
struct acct_t accounts[MAX_ACCT];     // current balances

struct trans_t { int id; int amount; };
struct trans_t transactions[MAX_TRANS];  // debit amounts

for (i = 0; i < MAX_TRANS; i++) {
  int id = transactions[i].id;
  int amount = transactions[i].amount;
  if (accounts[id].balance >= amount)
  {
    accounts[id].balance -= amount;
  }
}
```

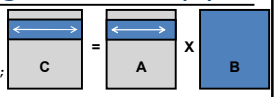
Can we do these in parallel?

7

7

Example: Parallelizing Matrix Multiply

```
for (I = 0; I < 100; I++)
  for (J = 0; J < 100; J++)
    for (K = 0; K < 100; K++)
      C[I][J] += A[I][K] * B[K][J];
```



How to parallelize matrix multiply?

- Replace outer "for" loop with "**parallel_for**"
- Support by many parallel programming environments

Implementation: give each of N processors loop iterations

```
int start = (100/N) * my_id();
for (I = start; I < start + 100/N; I++)
  for (J = 0; J < 100; J++)
    for (K = 0; K < 100; K++)
      C[I][J] += A[I][K] * B[K][J];
```

Each processor runs copy of loop above

- Library provides `my_id()` function

8

8

Example: Bank Accounts

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id, amt;
if (accts[id].bal >= amt)
{
   accts[id].bal -= amt;
}
```

```
r1 = id
r2 = amt
r3 = entry addr
r4 = bal
0: addi r3←r1,accts
1: lw  r4,0(r3)
2: blt r4,r2,done
3: sub r4←r4,r2
4: sw  0(r3),r4
```

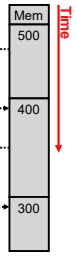
- Example of **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared "loosely" (sometimes yes, mostly no), dynamically
- Example: database/web server (each query is a thread)
  - `accts` is **shared**, can't register allocate even if it were scalar
  - `id` and `amt` are private variables, register allocated to `r1`, `r2`
- Running example

9

9

An Example Execution

```
      Thread 0                  Thread 1
0: addi r3←r1,accts
1: lw  r4,0(r3)
2: blt r4,r2,done
3: sub r4←r4,r2
4: sw  0(r3),r4
                          0: addi r3←r1,accts
                          1: lw  r4,0(r3)
                          2: blt r4,r2,done
                          3: sub r4←r4,r2
                          4: sw  0(r3),r4
```

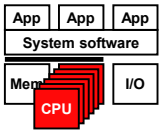


- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction executed on different processor
  - Track `accts[241].bal` (address is in `r3`)

10

10

Roadmap Checkpoint



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models

11

11

First, Uniprocessor Concurrency

- Software "thread"
  - Independent flow of execution
  - Context state: PC, registers
  - Threads generally share the same memory space
  - "Process" like a thread, but different memory space
  - Java has thread support built in, C/C++ supports P-threads library
- Generally, system software (the OS) manages threads
  - "Thread scheduling", "context switching"
  - All threads share the one processor
    - Hardware timer interrupt occasionally triggers OS
    - Quickly swapping threads gives illusion of concurrent execution
  - Much more in an operating systems course

12

12

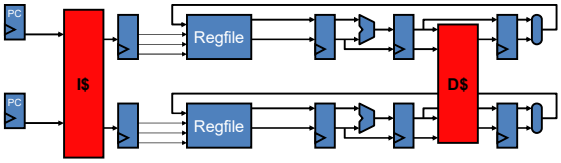
Multithreaded Programming Model

- Programmer explicitly creates multiple threads
- All loads & stores to a single **shared memory** space
  - Each thread has a private stack frame for local variables
- A "thread switch" can occur at any time
  - Pre-emptive multithreading by OS
- Common uses:
  - Handling user interaction (GUI programming)
  - Handling I/O latency (send network message, wait for response)
  - Expressing parallel work via Thread-Level Parallelism (TLP)

Shared Memory Implementations

- **Multiplexed uniprocessor**
  - Runtime system and/or OS occasionally pre-empt & swap threads
  - Interleaved, but no parallelism
- **Hardware multithreading** (previous unit)
  - Tolerate pipeline latencies, higher efficiency
  - Same interleaved shared-memory model
- **Multiprocessing**
  - Multiply execution resources, higher peak performance
  - Same interleaved shared-memory model
  - Foreshadowing: allow private caches, further disentangle cores
- **All support the shared memory programming model**

Simplest Multiprocessor

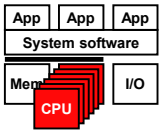


- Replicate entire processor pipeline!
  - Instead of replicating just register file & PC
  - Exception: share caches (we'll address this bottleneck later)
- Same "shared memory" or "multithreaded" model
  - Loads and stores from two processors are interleaved
- Advantages/disadvantages over hardware multithreading?

Shared Memory Issues

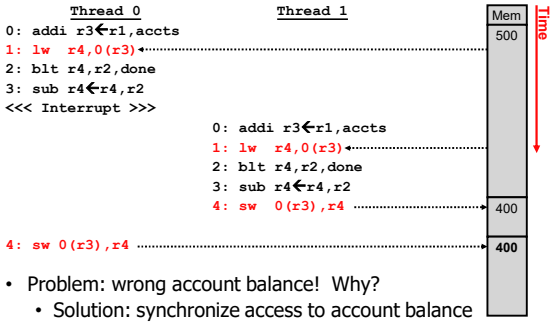
- Three in particular, not unrelated to each other
- Synchronization
  - How to regulate access to shared data?
  - How to implement critical sections?
- Cache coherence
  - How to make writes to one cache "show up" in others?
- Memory consistency model
  - How to keep programmer sane & let hw optimize?
  - How to reconcile shared memory with store buffers?

Roadmap Checkpoint



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multihreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models

A **Problem** Execution



- Problem: wrong account balance! Why?
  - Solution: synchronize access to account balance

Synchronization:

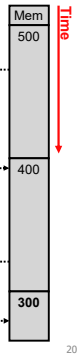
Synchronization: a key issue for shared memory
- Regulate access to shared data (mutual exclusion)
- Low-level primitive: **lock** (higher-level: "semaphore" or "mutex")
  - Operations: `acquire(lock)` and `release(lock)`
  - Region between `acquire` and `release` is a **critical section**
  - Must interleave `acquire` and `release`
  - Interfering `acquire` will block
- Another option: **Barrier synchronization**
  - Blocks until all threads reach barrier, used at end of "parallel_for"

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
shared int lock;
int id, amt;
acquire(lock);                          critical section
if (accts[id].bal >= amt) {
   accts[id].bal -= amt;
}
release(lock);
```


19

A Synchronized Execution

```
        Thread 0                    Thread 1
   call acquire(lock)
0: addi r3←r1,accts
1: lw  r4,0(r3)
2: blt r4,r2,done
3: sub r4←r4,r2
<<< Interrupt >>>          call acquire(lock) Spins!
                           <<< Interrupt >>>
4: st  0(r3),r4
   call release(lock)
                           (still in acquire,
                            which now returns)
                           0: addi r3←r1,accts
                           1: lw  r4,0(r3)
                           2: blt r4,r2,done
                           3: sub r4←r4,r2
                           4: sw  0(r3),r4
                              call release(lock)
```



Fixed, but how do we implement acquire & release?

20

Strawman Lock (Incorrect)

Spin lock: software lock implementation

- `acquire(lock):` `while (lock != 0) {}`
  `lock = 1;`

  "Spin" while lock is 1, wait for it to turn 0

```
A0:  lw r6,0(&lock)
A1:  bne  r6,r0,A0
A2:  addi r6←r6,1
A3:  sw 0(&lock),r6
```

- `release(lock):` `lock = 0;`

```
R0:  sw 0(&lock),r0     // r0 holds 0
```


21

Strawman Lock (Incorrect)

```
        Thread 0                    Thread 1
A0: lw r6,0(&lock)
A1: bne  r6,r0,A0          A0: lw r6,0(&lock)
A2: addi r6←r6,1           A1: bne  r6,r0,A0
A3: sw 0(&lock),r6         A2: addi r6←r6,1
CRITICAL_SECTION           A3: sw 0(&lock),r6
                           CRITICAL_SECTION
```



- Spin lock makes intuitive sense, but doesn't actually work
  - Loads/stores of two `acquire` sequences can be interleaved
  - Lock `acquire` sequence also not atomic
  - **Same problem as before!**
- Note, `release` is trivially atomic

22

A Correct Implementation: SYSCALL Lock

```
ACQUIRE_LOCK:
A1: disable_interrupts                  atomic
A2: lw r6,0(&lock),r6
A3: bne  r6,r0,A0
A4: addi r6←r6,1
A5: sw 0(&lock),r6
A6: enable_interrupts
A7: return
```

Implement lock in a SYSCALL
- Only kernel can control interleaving by disabling interrupts

\+ Works…

– Large system call overhead

– But not in hardware multithreading or a multiprocessor…

23

Better Spin Lock: Use Atomic Swap

- ISA provides an atomic lock acquisition instruction
  - Example: **atomic swap**

    `swap r1,0(&lock)`
  - Atomically executes:

```
mov tmp←r1
ld r1,0(&lock)
st 0(&lock),tmp
```

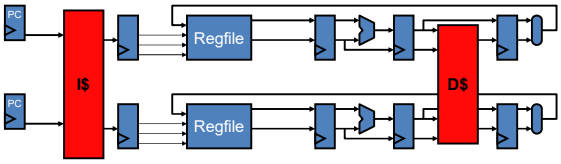
- New acquire sequence

```
(value of r1 is 1)
A0: swap r1,0(&lock)
A1: bne  r1,r0,A0
```

  - If lock was initially busy (1), doesn't change it, **keep looping**
  - If lock was initially free (0), acquires it (sets it to 1), break loop
- Ensures lock held by **at most one thread**
  - Other variants: **exchange**, **compare-and-swap**, **test-and-set (t&s)**, or **fetch-and-add**

24

Atomic Update/Swap Implementation



- How is atomic swap implemented?
  - Need to ensure no intervening memory operations
  - Requires blocking access by other threads temporarily (yuck)
- How to pipeline it?
  - Both a load and a store (yuck)
  - Not very RISC-like
  - Some ISAs provide a "load-link" and "store-conditional" insn pair

RISC Test-And-Set

- **`swap`**: a load and store in one insn is not very "RISC"
  - Broken up into micro-ops, but then how is it made atomic?
- **`ll`/`sc`**: load-locked / store-conditional
  - Atomic load/store pair

```
ll r1,0(&lock)
// potentially other insns
sc 0(&lock),r2
```

  - On **`ll`**, processor remembers address…
    - …And looks for writes by other processors
    - If write is detected, next **`sc`** to same address is annulled
      - Sets failure condition
- On RISC-V ISA called load-reserved/store-conditional

Lock Correctness

```
      Thread 0              Thread 1
A0: swap r1,0(&lock)
A1: bne  r1,r0,A0     A0: swap r1,0(&lock)
CRITICAL_SECTION      A1: bne  r1,r0,A0
                      A0: swap r1,0(&lock)
                      A1: bne  r1,r0,A0
```

+ Lock actually works…
  - Thread 1 keeps spinning

- Sometimes called a "test-and-set lock"
  - Named after the common "test-and-set" atomic instruction

"Test-and-Set" Lock Performance

```
Thread 0              Thread 1
A0: swap r1,0(&lock)
A1: bne  r1,r0,A0     A0: swap r1,0(&lock)
A0: swap r1,0(&lock)  A1: bne  r1,r0,A0
A1: bne  r1,r0,A0     A0: swap r1,0(&lock)
                      A1: bne  r1,r0,A0
```

– …but performs poorly
- Consider 3 processors rather than 2
- P2 (not shown) has the lock and is in the critical section
- But what are P0 and P1 doing in the meantime?
  - Loops of **`swap`**, each of which includes a **`store`**
  - – Repeated stores by multiple processors costly (more in a bit)
  - – Generating a ton of useless interconnect traffic

Test-and-Test-and-Set Locks

Solution: **test-and-test-and-set locks**

- New acquire sequence

```
A0: ld   r1,0(&lock)
A1: bne  r1,r0,A0
A2: addi r1←r1,1
A3: swap r1,0(&lock)
A4: bne  r1,r0,A0
```

- Within each loop iteration, before doing a **`swap`**
  - Spin doing a simple test (**`ld`**) to see if lock value has changed
  - Only do a **`swap`** (**`st`**) if lock is actually free
- Processors can spin on a busy lock locally (in their own cache)
  - + Less unnecessary interconnect traffic
- Note: test-and-test-and-set is *not* a new instruction!
  - Just different software

Queue Locks

- Test-and-test-and-set locks can still perform poorly
  - If lock is contended for by many processors
  - Lock release by one processor, creates "free-for-all" by others
  - – Interconnect gets swamped with **`t&s`** requests
- **Software queue lock**
  - Each waiting processor spins on a different location (a queue)
  - When lock is released by one processor...
    - Only the next processors sees its location go "unlocked"
    - Others continue spinning locally, unaware lock was released
  - Effectively, passes lock from one processor to the next, in order
  - + Greatly reduced network traffic (no mad rush for the lock)
  - + Fairness (lock acquired in FIFO order)
  - – Higher overhead in case of no contention (more instructions)
  - – Poor performance if one thread gets swapped out

Programming With Locks Is Tricky

- Multicore processors are the way of the foreseeable future
  - thread-level parallelism anointed as parallelism model of choice
  - Just one problem…
- Writing lock-based multi-threaded programs is tricky!
- More precisely:
  - Writing programs that are correct is "easy" (not really)
  - Writing programs that are highly parallel is "easy" (not really)
  - *Writing programs that are both correct and parallel is difficult*
    - And that's the whole point, unfortunately
  - Selecting the "right" kind of lock for performance
    - Spin lock, queue lock, ticket lock, read/writer lock, *etc.*
  - **Locking granularity issues**

Goldibear and the 3 Locks

- **Coarse-grain locks**: **correct, but slow**
  - one lock for entire database
  - \+ Easy to make correct: no chance for unintended interference
  - – Limits parallelism: no two critical sections can proceed in parallel
- **Fine-grain locks**: **parallel, but difficult**
  - multiple locks, one per record
  - \+ Fast: critical sections (to different records) can proceed in parallel
  - – Difficult to make correct: easy to make mistakes
- **Multiple locks**: **just right?** (sorry, no fairytale ending)
  - acct-to-acct transfer: must acquire both `id_from`, `id_to` locks
  - Simultaneous transfers 241 → 37 and 37 → 241
  - **Deadlock**: circular wait for shared resources
  - **Solution:** Always acquire multiple locks in same order
    - Just another thing to keep in mind when programming

More Lock Madness

- What if…
  - Some actions (*e.g.*, deposits, transfers) require 1 or 2 locks…
  - …and others (*e.g.*, prepare statements) require all of them?
  - Can these proceed in parallel?
- What if…
  - There are locks for global variables (*e.g.*, operation id counter)?
  - When should operations grab this lock?
- What if… what if… what if…

Lock-based programming is difficult…
…wait, it gets worse

And To Make It Worse…

- **Acquiring locks is expensive…**
  - By definition requires a slow atomic instructions
    - Specifically, acquiring write permissions to the lock
  - Ordering constraints (see soon) make it even slower
- **…and 99% of the time un-necessary**
  - Most concurrent actions don't actually share data
  - – You paying to acquire the lock(s) for no reason
- Fixing these problem is an area of active research
  - One proposed solution "Transactional Memory"

Research: Transactional Memory (TM)

Transactional Memory

\+ Programming simplicity of coarse-grain locks
\+ Higher concurrency (parallelism) of fine-grain locks
  - Critical sections only serialized if data is actually shared
\+ No lock acquisition overhead

- Hottest thing since sliced bread (or was a few years ago)
- No fewer than eight research projects:
  - Brown, Stanford, MIT, Wisconsin, Texas, Rochester, Intel, Penn

Transactional Memory: The Big Idea

- Big idea I: **no locks, just shared data**
  - "Look ma, no locks"
- Big idea II: **optimistic (speculative) concurrency**
  - Execute critical section speculatively, abort on conflicts
  - Better to beg for forgiveness than to ask for permission
- **Read set**: set of shared addresses critical section reads
  - Example: `accts[37].bal`, `accts[241].bal`
- **Write set**: set of shared addresses critical section writes
  - Example: `accts[37].bal`, `accts[241].bal`

Transactional Memory: Begin

`begin_transaction`

- Take a local register checkpoint
- Begin locally tracking read set (remember addresses you read)
  - See if anyone else is trying to write it
- Locally buffer all of your writes (invisible to other processors)

+ **Local actions only: no lock acquire**

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```


Transactional Memory: End

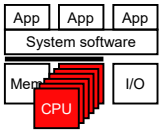
`end_transaction`

- Check read set: is data you read still valid (*i.e.*, no writes to any)
- Yes? Commit transactions: commit writes
- No? Abort transaction: restore checkpoint

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

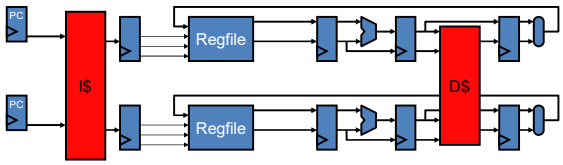
begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```


Roadmap Checkpoint



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models

Recall: Simplest Multiprocessor

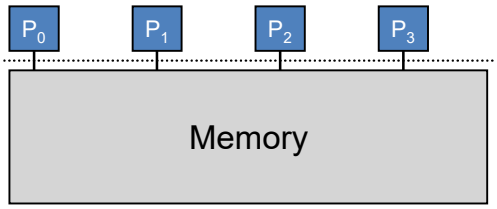


- What if we don't want to share the L1 caches?
  - Bandwidth and latency issue
- Solution: use per-processor ("private") caches
  - Coordinate them with a ***Cache Coherence Protocol***

Shared-Memory Multiprocessors

Conceptual model

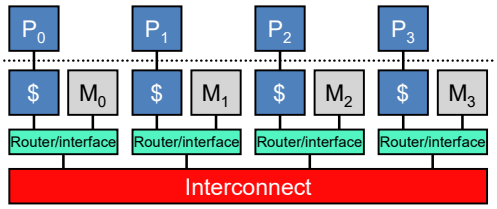
- The shared-memory abstraction
- Familiar and feels natural to programmers
- Life would be easy if systems actually looked like this…



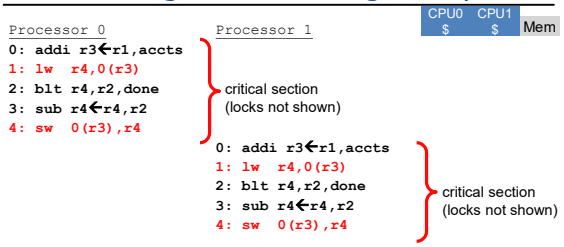
Shared-Memory Multiprocessors

…but systems actually look more like this

- Processors have caches
- Memory may be physically distributed
- Arbitrary interconnect



Revisiting Our Motivating Example

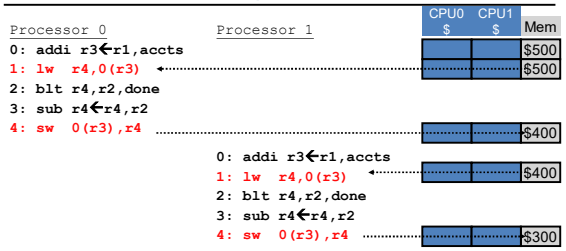


- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `r3`)

43

43

No-Cache, No-Problem

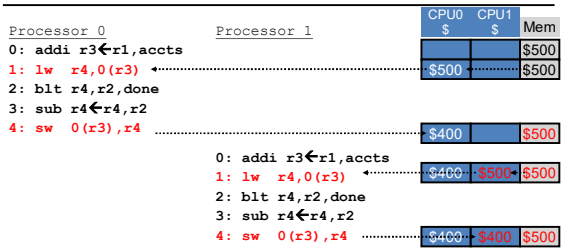


- Scenario I: processors have no caches
  - No problem

44

44

Cache Incoherence

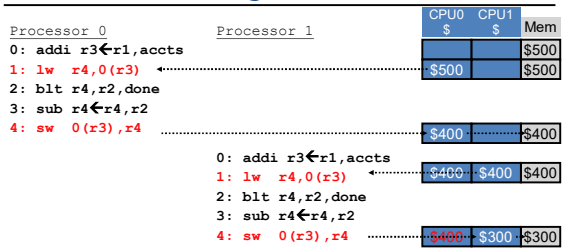


- Scenario II(a): processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
  - Can get incoherent (inconsistent)

45

45

Write-Through Doesn't Fix It



- Scenario II(b): processors have write-through caches
  - This time only 2 (different) copies of `accts[241].bal`
  - No problem? What if another withdrawal happens on processor 0?

46

46

What To Do?

- No caches?
  - Slow
- Make shared data uncachable?
  - Faster, but still too slow
  - Entire `accts` database is technically "shared"
- Flush all other caches on writes to shared data?
  - May as well not have caches
- **Hardware cache coherence**
  - Rough goal: all caches have same data at all times
  + Minimal flushing, maximum caching → best performance

47

47

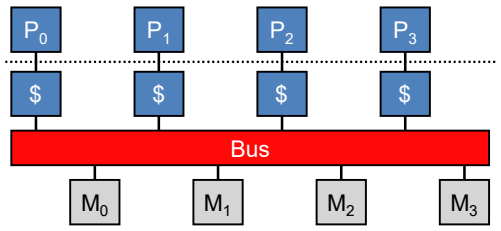
Bus-based Multiprocessor

Simple multiprocessors use a bus

- **All** processors see **all requests** at the **same time**, same order

Memory

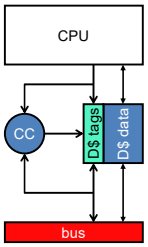
- Single memory module, **-or-**
- Banked memory module



48

48

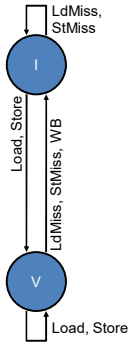
Hardware Cache Coherence



- **Coherence**
  - all copies have same data at all times
- **Coherence controller**:
  - Examines bus traffic (addresses and data)
  - Executes **coherence protocol**
    - What to do with local copy when you see different things happening on bus
- Three processor-initiated events
  - **Ld**: load
  - **St**: store
  - **WB**: write-back
- Two remote-initiated events
  - **LdMiss**: read miss from ***another*** processor
  - **StMiss**: write miss from ***another*** processor

49

VI (MI) Coherence Protocol



- **VI (valid-invalid) protocol**: aka MI*
  - Two states (per block in cache)
    - **V (valid)**: have block
    - **I (invalid)**: don't have block
    - +Can implement with valid bit
  - Protocol diagram (left)
    - If *you* load/store a block: transition to **V**
    - If anyone *else* wants to read/write block:
      - Give it up: transition to **I** state
      - Write-back if your own copy is dirty
  - This is an **invalidate protocol**
  - **Update protocol**: copy data, don't invalidate
    - Sounds good, but wastes a lot of bandwidth

* M=modified, comes later

50

VI Protocol (Write-Back Cache)

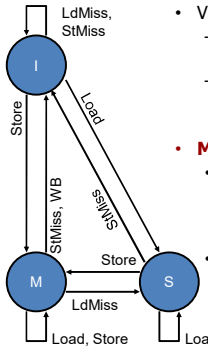
```
Processor 0                 Processor 1
0: addi r3←r1,accts
1: lw  r4,0(r3)
2: blt r4,r2,done
3: sub r4←r4,r2
4: sw  0(r3),r4
                            0: addi r3←r1,accts
                            1: lw  r4,0(r3)
                            2: blt r4,r2,done
                            3: sub r4←r4,r2
                            4: sw  0(r3),r4
```

CPU0	CPU1	Mem
		500
V:500		500
V:400		500
I:	V:400	400
	V:300	400

`load` by processor 1 generates an "other load miss" event (LdMiss)
- processor 0 responds by sending its dirty copy, transitioning to **I**

51

VI → MSI



- VI protocol is inefficient
  - Only one cached copy allowed in entire system
  - Multiple copies can't exist even if read-only
    - Not a problem in example
    - Big problem in reality
- **MSI (modified-shared-invalid)**
  - Fixes problem: splits "V" state into two states
    - **M (modified)**: local dirty copy
    - **S (shared)**: local clean copy
  - Allows **either**
    - Multiple read-only copies (S-state) **--OR--**
    - Single read/write copy (M-state)

52

MSI Protocol (Write-Back Cache)

```
Processor 0                 Processor 1
0: addi r3←r1,accts
1: lw  r4,0(r3)
2: blt r4,r2,done
3: sub r4←r4,r2
4: sw  0(r3),r4
                            0: addi r3←r1,accts
                            1: lw  r4,0(r3)
                            2: blt r4,r2,done
                            3: sub r4←r4,r2
                            4: sw  0(r3),r4
```

CPU0	CPU1	Mem
		500
S:500		500
M:400		500
S:400	S:400	400
I:	M:300	400

`ld` by processor 1 generates a "other load miss" event (LdMiss)
- Processor 0 responds by sending its dirty copy, transitioning to **S**

`st` by processor 1 generates a "other store miss" event (StMiss)
- Processor 0 responds by transitioning to **I**

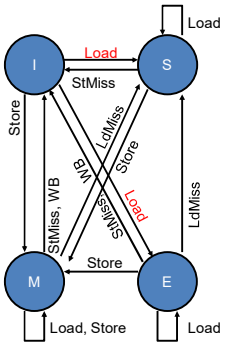
53

Cache Coherence and Cache Misses

- Coherence introduces two new kinds of cache misses
  - **Upgrade miss**
    - On stores to read-only blocks
    - Delay to acquire write permission to read-only block
  - **Coherence miss**
    - Miss to a block evicted by another processor's requests
- Making the cache larger…
  - Doesn't reduce these type of misses
  - As cache grows large, these sorts of misses dominate
- **False sharing**
  - Two or more processors sharing parts of the same block
  - But *not* the same bytes within that block (no actual sharing)
  - Creates pathological "ping-pong" behavior
  - Careful data placement may help, but is difficult

54

MSI → MESI: *Exclusive Clean* Protocol



Most modern protocols also include **E (exclusive)** state

- "I have the only cached copy, and it's a **clean** copy"
- Why is this state useful?

Load transitions to E if no other processors is caching the block, otherwise S

55

55

Exclusive Clean Protocol Optimization

```
Processor 0                Processor 1
0: addi r3←r1,accts
1: lw  r4,0(r3)
2: blt r4,r2,done
3: sub r4←r4,r2
4: sw  0(r3),r4
```

	CPU0	CPU1	Mem
			500
E:	500		500
(No miss)	M:	400	500
	S:	400	S:	400	400
	I:		M:	300	400

```
0: addi r3←r1,accts
1: lw  r4,0(r3)
2: blt r4,r2,done
3: sub r4←r4,r2
4: sw  0(r3),r4
```

56

56

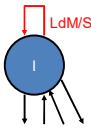
Snooping Bandwidth Scaling Problems

- Coherence events generated on…
  - L2 misses (and writebacks) – *actually* last level cache misses
- Problem#1: **N^2 bus traffic**
  - All N processors send their misses to all N-1 other processors
  - Assume: 2 IPC, 2 GHz clock, 0.01 misses/insn **per processor**
  - 0.01 misses/insn x 2 insn/cycle x 2 cycle/ns x 64 B blocks
    = 2.56 GB/s… per processor
    - With 16 processors, that's 40 GB/s! With 128 that's 320 GB/s!!
  - You can use multiple buses… but that hinders global ordering
- Problem#2: **N^2 processor snooping bandwidth**
  - 0.01 events/insn x 2 insn/cycle = 0.02 events/cycle per processor
  - 16 processors: 0.32 bus-side tag lookups per cycle
    - Add 1 extra port to cache tags? Okay
  - 128 processors: 2.56 tag lookups per cycle! 3 extra tag ports?

57

57

"Scalable" Cache Coherence



Part I: **bus bandwidth**

Replace non-scalable bandwidth substrate (bus)…
…with scalable one (point-to-point network, *e.g.*, mesh)

Part II: **processor snooping bandwidth**

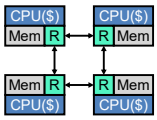
- Most snoops result in no action
- Replace non-scalable broadcast protocol (spam everyone)…

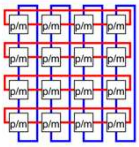
…with scalable **directory protocol** (only notify processors that care)

58

58

Scalable Cache Coherence





- Point-to-point interconnects
  - **Glueless MP**: no need for additional "glue" chips
  - + Can be arbitrarily large: 1000's of processors
    - **Massively parallel processors (MPPs)**
  - AMD Opteron/Phenom – point-to-point, glueless, broadcast
- Distributed memory: non-uniform memory architecture (NUMA)
  - AMD EPYC – L3 distributed across chiplets
- Multicore: on-chip interconnection networks

59

59

Directory Coherence Protocols

Observe: address space statically partitioned
+ Can easily determine which memory module holds a given line
  - That memory module sometimes called "**home**"
– Can't easily determine which processors have line in their caches
- Bus-based protocol: broadcast events to all processors/caches
  ± Simple and fast, but non-scalable

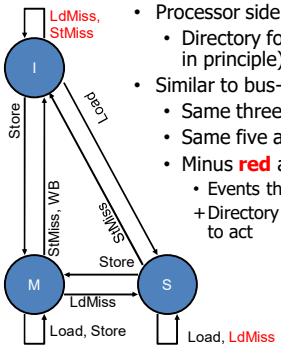
Directories: non-broadcast coherence protocol

- Extend memory to track caching information
- For each physical cache line whose home this is, track:
  - **Owner**: which processor has a dirty copy (*i.e.*, M state)
  - **Sharers**: which processors have clean copies (*i.e.*, S state)
- Processor sends coherence event to home directory
  - Home directory only sends events to processors that care
- For multicore w/ shared L3, put directory info in cache tags

60

60

MSI Directory Protocol



- Processor side
  - Directory follows its own protocol (obvious in principle)
- Similar to bus-based MSI
  - Same three states
  - Same five actions (keep BR/BW names)
  - Minus **red** arcs/actions
    - Events that would not trigger action anyway
    - \+ Directory won't bother you unless you need to act

61

MSI Directory Protocol

```
Processor 0
0: addi r3←r1,accts
1: lw  r4,0(r3)
2: blt r4,r2,done
3: sub r4←r4,r2
4: sw  0(r3),r4
```

```
Processor 1
0: addi r3←r1,accts
1: lw  r4,0(r3)
2: blt r4,r2,done
3: sub r4←r4,r2
4: sw  0(r3),r4
```

P0	P1	Directory
		–:–:500
S:500		S:0:500
M:400		M:0:500 (stale)
S:400	S:400	S:0,1:400
I:	M:300	M:1:400

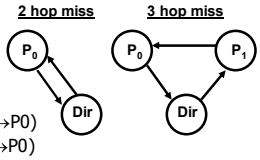
`lw` by P1 sends BR to directory
- Directory sends BR to P0, P0 sends P1 data, does WB, goes to **S**

`sw` by P1 sends BW to directory
- Directory sends BW to P0, P0 goes to **I**

62

Directory Flip Side: Latency

- Directory protocols
  - \+ Lower bandwidth consumption → more scalable
  - – Longer latencies
- Two read miss situations



- Unshared: get data from memory
  - Snooping: 2 hops (P0→memory→P0)
  - Directory: 2 hops (P0→memory→P0)
- Shared or exclusive: get data from other processor (P1)
  - Assume cache-to-cache transfer optimization
  - Snooping: 2 hops (P0→P1→P0)
  - – Directory: **3 hops** (P0→memory→P1→P0)
  - Common, many processors → high probability someone has it

63

Directory Flip Side: Complexity

- Latency is not the only issue for directories
  - Subtle correctness issues as well
  - Stem from unordered nature of underlying inter-connect
- Individual requests to single cache must be ordered
  - Bus-based snooping: all processors see all requests in same order
    - Ordering automatic
  - Point-to-point network: requests may arrive in different orders
    - Directory has to enforce ordering explicitly
    - Cannot initiate actions on request B…
      …until all relevant processors complete actions on request A
    - Requires directory to collect acks, queue requests, *etc.*
- Directory protocols
  - Obvious in principle
  - – Complicated in practice

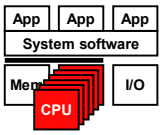
64

Coherence on Real Machines

- Many uniprocessors designed with on-chip snooping logic
  - Can be easily combined to form multi-processors
    - *e.g.*, Intel Pentium4 Xeon
  - And multicore, of course
- Larger scale (directory) systems built from smaller MPs
  - *e.g.*, Sun Wildfire, NUMA-Q, IBM Summit
- Some shared memory machines are **not cache coherent**
  - *e.g.*, CRAY-T3D/E
  - Shared data is uncachable
  - If you want to cache shared data, copy it to private data section
  - Basically, cache coherence implemented in software
    - Have to really know what you are doing as a programmer

65

Roadmap Checkpoint



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- **Memory consistency models**

66

Tricky Shared Memory Examples

- Answer the following questions:
  - **Initially: all variables zero** (that is, x is 0, y is 0, flag is 0, A is 0)
  - What value pairs can be read by the two loads? (x, y) pairs:

thread 1	thread 2
`load x`	`store y ← 1`
`load y`	`store x ← 1`

  - (0,0) and (1,1) easy to see
  - `load x, store y ← 1, load y, store x ← 1` gives (0,1)
  - Is it possible to get (1,0)?

Tricky Shared Memory Examples

- Answer the following questions:
  - **Initially: all variables zero** (that is, x is 0, y is 0, flag is 0, A is 0)
  - What value pairs can be read by the two loads? (x, y) pairs:

thread 1	thread 2
`load x`	`store y ← 1`
`load y`	`store x ← 1`

  - What value pairs can be read by the two loads? (x, y) pairs:

thread 1	thread 2
`store y ← 1`	`store x ← 1`
`load x`	`load y`

  - What value can be read by "Load A" below?

thread 1	thread 2
`store A ← 1`	`while(flag == 0) { }`
`store flag ← 1`	`load A`

Memory Consistency

- **Memory coherence**
  - Creates globally uniform (consistent) view…
    …of **a single memory location** (in other words: cache line)
  - – Not enough
    - Cache lines A and B can be individually consistent…
      …but inconsistent *with respect to each other*
- **Memory consistency**
  - Creates globally uniform (consistent) view…
    …of **all memory locations relative to each other**
- Who cares? Programmers
  - – Globally inconsistent memory creates mystifying behavior

Hiding Store Miss Latency

- Recall (back from caching unit)
  - Hiding store miss latency
  - How? Store buffer
- Said it would complicate multiprocessors
  - Yes, it does!

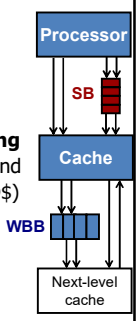
Write Misses and Store Buffers

Read miss?
- Load can't go on without the data→must stall

Write miss?
- Technically, no one needs data→why stall?

Store buffer: a small buffer
- Stores put addr/value to write buffer, **keep going**
- Store buffer writes stores to D$ in the background
- Loads must search store buffer (in addition to D$)

+ Eliminates stalls on write misses (mostly)
– Creates some problems



Store Buffers & Consistency

```
          A=flag=0;
Processor 0          Processor 1
A=1;                 while (!flag); // spin
flag=1;              print A;
```

Consider the following execution:
- Processor 0's write to A, misses the cache. Put in store buffer
- Processor 0 keeps going
- Processor 0 write "1" to flag hits, completes
- Processor 1 reads flag… sees the value "1"
- Processor 1 exits loop
- Processor 1 prints "0" for A

Ramification: store buffers can cause "strange" behavior
- How strange depends on lots of things

Coherence vs. Consistency

```
          A=0  flag=0
Processor 0          Processor 1
A=1;                 while (!flag); // spin
flag=1;              print A;
```

- **Intuition says**: P1 prints A=1
- **Coherence says**: absolutely nothing
  - P1 can see P0's write of `flag` before write of `A`!!! How?
    - **P0 has a coalescing store buffer that reorders writes**
    - **Or out-of-order execution**
    - **Or compiler re-orders instructions**
- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't
- **Real systems** act in this strange manner
  - What is allowed is defined as part of the ISA of the processor

Memory Consistency Models

- **Sequential consistency (SC)** (MIPS, PA-RISC)
  - **Formal definition of memory view programmers expect**
  - Processors see their own loads and stores in program order
    - + Provided naturally, even with out-of-order execution
  - But also: processors see others' loads and stores in program order
  - And finally: all processors see same global load/store ordering
    - – Last two conditions not naturally enforced by coherence
  - Corresponds to some sequential interleaving of uniprocessor orders
  - **Indistinguishable from multi-programmed uni-processor**
- **Processor consistency (PC)** (x86, SPARC)
  - Also called total store ordering (TSO)
  - Allows a in-order store buffer
    - Stores can be deferred, but must be put into the cache in order
- **Release consistency (RC)** (ARM, Itanium, PowerPC, RISC-V)
  - Allows an un-ordered store buffer
    - Stores can be put into cache in any order

Restoring Order

- Sometimes we need ordering (mostly we don't)
  - Prime example: ordering between "lock" and data
- How? insert **Fences (memory barriers)**
  - Special instructions, part of ISA
- Example
  - Ensure that loads/stores don't cross lock acquire/release operation

```
acquire
fence
critical section
fence
release
```

- How do fences work?
  - They stall execution until write buffers are empty
  - Makes lock acquisition and release slow(er)
- **Use synchronization library, don't write your own**

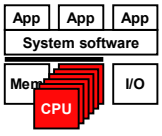
Shared Memory Summary

- **Synchronization**: regulated access to shared data
  - Key feature: atomic lock acquisition operation (e.g., `t&s`)
  - Performance optimizations: test-and-test-and-set, queue locks
- **Coherence**: consistent view of individual cache lines
  - Absolute coherence not needed, relative coherence OK
  - VI and MSI protocols, cache-to-cache transfer optimization
  - Implementation? snooping, directories
- **Consistency**: consistent view of all memory locations
  - Programmers intuitively expect sequential consistency (SC)
    - Global interleaving of individual processor access streams
    - – Not always naturally provided, may prevent optimizations
  - Weaker ordering: consistency only for synchronization points

Flynn's Taxonomy

- Proposed by Michael Flynn in 1966
- SISD – single instruction, single data
  - Traditional uniprocessor
- SIMD – single instruction, multiple data
  - Execute the same instruction on many data elements
  - Vector machines, graphics engines
- MIMD – multiple instruction, multiple data
  - Each processor executes its own instructions
  - Multicores are all built this way
  - SPMD – single program, multiple data (extension proposed by Frederica Darema)
    - MIMD machine, each node is executing the same code
- MISD – multiple instruction, single data
  - Systolic array

Summary



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models